```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

#ifndef DSF_H
#define DSF_H
#include <iostream>
#include <stdlib.h>

using namespace std;

template<class T>
class DSNode
{
public:
  DSNode(); // default constructor
  DSNode(T* initKey); // constructor
  DSNode(T* initKey, T* initParent); // constructor
  DSNode(T* initKey, T* initParent, int initRank); // constructor
  int rank; // rank of each node
  DSNode<T>* parent; // node pointer to parent node
  T* key;
};

template<class T>
class DisjointSets
{
public:
  DisjointSets(); // default constructor
  DisjointSets(int size); // constructor with given capacity
  DisjointSets(const DisjointSets<T>& ds); // copy constructor
  ~DisjointSets(); // destructor

  DSNode<T>* makeSet(T* x); // make a new singleton set containing data x
  void unionSets(DSNode<T>* x, DSNode<T>* y); // union the disjoint sets containing data
x and y
  DSNode<T>* findSet(DSNode<T>* x); // return the representative of the set containing x

  DisjointSets<T>& operator=(const DisjointSets<T>& ds); // assignment operator

  std::string toString(); // return a string representation of the disjoint set forest

private:
  DSNode<T>* copy(const DSNode<T>* node); // copy helper function
  void link(DSNode<T>* x, DSNode<T>* y); // link two disjoint sets together

  DSNode<T> **elements; // array of nodes in the forest
  int capacity; // size of elements array
  int length; // number of elements in the forest
};

template<class T>
std::ostream& operator<<(std::ostream& stream, const DisjointSets<T>& ds); // ostream ope
rator

class FullErr { }; // full exception
class NotFoundError { }; // element not found exception

#include "DSF.cpp"

#endif
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

#include <iostream>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sstream>
#include <math.h>
#include "list.h"

using namespace std;

/*=======================================================
Node Default Constructor
Precondition: None
Postcondition: Initializes empty DSNode
=====================================================*/
template<class T>
DSNode<T>::DSNode()
{
  key = NULL;
  parent = NULL;
  rank = 0;
}

/*=========================================================
Node Constructor
Precondition: initKey is a pointer to type T
Postcondition: Creates DSNode with rank 0 and key initKey
=======================================================*/
template<class T>
DSNode<T>::DSNode(T* initKey)
{
  key = initKey;
  parent = NULL;
  rank = 0;
}

/*============================================================================
Node Constructor
Precondition: initKey and initParent are pointers to type T
Postcondition: Creates DSNode with rkey initKey, parent initParent, and rank 0
==========================================================================*/
template<class T>
DSNode<T>::DSNode(T* initKey, T* initParent)
{
  key = initKey;
  parent = initParent;
  rank = 0;
}

/*===================================================================================
Node Constructor
Precondition: initKey and initParent are pointers to type T, initRank is an integer
Postcondition: Creates DSNode with key initKey, parent initParent, and rank initRank
=================================================================================*/
template<class T>
DSNode<T>::DSNode(T* initKey, T* initParent, int initRank)
{
  key = initKey;
  parent = initParent;
  rank = initRank;
}

/*=======================================================
Default Constructor
Precondition: None
```

```
Postcondition: Creates empty DisjointSets
=======================================================*/
template<class T>
DisjointSets<T>::DisjointSets()
{
  elements = new DSNode<T>*[100]; // set default as 100
  length = 0;
  capacity = 100;
}

/*============================================================
Constructor with Capacity
Precondition: Size is an integer
Postcondition: Creates empty DisjointSets with capacity size
============================================================*/
template<class T>
DisjointSets<T>::DisjointSets(int size)
{
  elements = new DSNode<T>*[size];
  length = 0;
  capacity = size;
}

/*=======================================================
Copy Constructor
Precondition: ds is a DisjointSets of type T
Postcondition: Creates a copy of ds
=======================================================*/
template<class T>
DisjointSets<T>::DisjointSets (const DisjointSets<T>& ds)
{
  capacity = ds.capacity;
  elements = new DSNode<T>*[capacity];
  length = ds.length;

  for (int i = 0; i < length; i++)
  {
      elements[i] = copy(ds.elements[i]);
  }
}

/*=======================================================
copy Function
Precondition: node is a pointer to DSNode
Postcondition: returns a copy of node
=======================================================*/
template<class T>
DSNode<T>* DisjointSets<T>::copy (const DSNode<T>* node)
{
  DSNode<T> *newNode = new DSNode<T>;
  if(node == NULL)
  {
    return NULL;
  }

  newNode->rank = node->rank;
  newNode->key = node->key;
  if(node->parent == node)
    newNode->parent = newNode;
  else
    newNode->parent = copy(node->parent);
  return newNode;
}

/*=======================================================
Destructor
Precondition: None
Postcondition: Deallocates the memory of DisjointSets
=======================================================*/
template<class T>
```

```
DisjointSets<T>::~DisjointSets()
{
  List<T> alreadyDeleted;
  for(int i = 0; i < length; i++)
  {
    alreadyDeleted.append(elements[i]->key);
    bool found = false;
    for(int j = 0; j < alreadyDeleted.length(); j++)
    {
      if(elements[i]->key == alreadyDeleted[j])
      {
        found = true;
      }
    }
    if(found == false)
    {
        delete [] elements[i];
    }
  }
  length = 0;
  capacity = 0;
}

/*=========================================================
Assignment Operator
Precondition: ds is a DisjointSets object
Postcondition: Creates a copy of ds
=========================================================*/
template<class T>
DisjointSets<T>& DisjointSets<T>::operator= (const DisjointSets<T>& ds)
{
  if(this != &ds)
  {
    delete [] elements;
    capacity = ds.capacity;
    elements = new DSNode<T>*[capacity];
    length = ds.length;
    for(int i = 0; i < length; i++)
    {
      elements[i] = copy(ds.elements[i]);
    }
  }
  return *this;
}

/*================================================================================
MaketSet Function
Precondition: Elements array of disjoint sets must not be full, x must be of type T,
and a node with key x must not already be in the forest.
Postcondition: A new singleton set has been created in the disjoint set forest.
================================================================================*/
template<class T>
DSNode<T>* DisjointSets<T>::makeSet(T* x)
{
  // error: disjoint set forest is full
  if (length == capacity)
  {
    throw FullErr();
  }

  // create pointer to new node with a key of x
  DSNode<T> *xnode = new DSNode<T>(x);

  // make new node its own parent and set rank to 0
  xnode->parent = xnode;
  xnode->rank = 0;

  // insert node as a new singleton set in the elements array at the next available index
  elements[length] = xnode;
```

```
  // increment length of forest
  length++;

  return xnode;
}

/*=========================================================================
UnionSets Function
Precondition: x and y are both pointers to nodes in the disjoint set forest.
Postcondition: The disjoint sets that contained nodes x and y have been joined
together as one set.
=========================================================================*/
template<class T>
void DisjointSets<T>::unionSets(DSNode<T>* x, DSNode<T>* y)
{
  // call find set on x and y to find their set representatives and then call link
  // with these representatives to join the sets together
  link(findSet(x), findSet(y));
}

/*=========================================================================
Link Function
Precondition: x and y are pointers to the representative nodes of their sets.
Postcondition: The sets of x and y have been linked together into one set.
=========================================================================*/
template<class T>
void DisjointSets<T>::link(DSNode<T>* x, DSNode<T>* y)
{
  // if the rank of x is greater than the rank of y, x becomes the representative of
  // the new set
  if (x->rank > y->rank)
  {
    y->parent = x;
  }

  // if the rank of y is greater than or equal to the rank of x, y becomes the
  // representative of the new set
  else
  {
    x->parent = y;

    // if the ranks of x and y are the same, increment y’s rank
    if (x->rank == y->rank)
    {
      y->rank = y->rank + 1;
    }
  }
}

/*=========================================================================
findSet Function
Precondition: x is a node in the disjoint set forest.
Postcondition: a pointer to the representative node of the set x is returned.
=========================================================================*/
template<class T>
DSNode<T>* DisjointSets<T>::findSet(DSNode<T>* x)
{
  // error: node is not in forest
  if (x->parent == NULL)
    throw NotFoundError();

  // if we haven’t reached the representative of the set, call findSet on the current
  // node’s parent
  if (x->parent != x)
    x->parent = findSet(x->parent); // path compression
  return x->parent;
}

/*============================================================================
toString Function
```

```
Precondition: called by valid DisjointSets object.
Postcondition: string representation of disjoint set forest is returned with each line
representing a node in the forest. i:j represents a node where i is the key and j is the
rank. Parent relationships of i:j are denoted with an '->' and then the parent's node
representation.
=========================================================================================*/
template<class T>
std::string DisjointSets<T>::toString()
{
  stringstream s;

  for(int i = 0; i < length; i++)
  {
    DSNode<T>* x = elements[i];
    while(x->parent != x)
    {
      s << *(x->key) << ":" << x->rank << " -> ";
      x = x->parent;
    }
    s << *(x->key) << ":" << x->rank << "\n";
  }

  string str = s.str();
  return str;
}

/*=========================================================================================
Stream Operator
Precondition: ds is a DisjointSets object
Postcondition: prints a string representation of a DisjointSets object
=========================================================================================*/
template<class T>
std::ostream& operator<<(std::ostream& stream, const DisjointSets<T>& ds)
{
  stream << ds.toString();
  return stream;
}
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

#include "DSF.h"
#include <iostream>
#include <stdlib.h>
#include <assert.h>
#include <string>

void test_constructor()
{
  try
  {
    // default constructor
    DisjointSets<int> set;
    assert(set.toString() == "");

    // constructor with size
    DisjointSets<int> set2(7);
    assert(set2.toString() == "");

    int *a = new int(5);
    int *b = new int(7);
    int *c = new int(20);
    int *d = new int(11);
    int *e = new int(13);
    int *f = new int(22);
    int *g = new int(3);

    DSNode<int> *node1 = set.makeSet(a);
    DSNode<int> *node2 = set.makeSet(b);
    DSNode<int> *node3 = set.makeSet(c);
    DSNode<int> *node4 = set.makeSet(d);
    DSNode<int> *node5 = set.makeSet(e);
    DSNode<int> *node6 = set.makeSet(f);
    DSNode<int> *node7 = set.makeSet(g);

    assert(set.toString() == "5:0\n7:0\n20:0\n11:0\n13:0\n22:0\n3:0");

    DisjointSets<int> set3(set);
    assert(set3.toString() == "5:0\n7:0\n20:0\n11:0\n13:0\n22:0\n3:0");

    DisjointSets<int> set4 = set3;
    assert(set4.toString() == "5:0\n7:0\n20:0\n11:0\n13:0\n22:0\n3:0");
  }

  catch (FullErr exception)
  {
    cout << "Error: forest of disjoint sets is full" << endl;
  }

  catch (NotFoundError exception)
  {
    cout << "Error: node not found" << endl;
  }
}

void test_makeSet()
{
  try
  {
    int *a = new int(5);
    int *b = new int(7);
    int *c = new int(20);
    int *d = new int(11);
    int *e = new int(13);
    int *f = new int(22);
    int *g = new int(3);
```

```
    DisjointSets<int> set(10);
    DSNode<int> *node1 = set.makeSet(a);
    DSNode<int> *node2 = set.makeSet(b);
    DSNode<int> *node3 = set.makeSet(c);
    DSNode<int> *node4 = set.makeSet(d);
    DSNode<int> *node5 = set.makeSet(e);
    DSNode<int> *node6 = set.makeSet(f);
    DSNode<int> *node7 = set.makeSet(g);

    assert(set.toString() == "5:0\n7:0\n20:0\n11:0\n13:0\n22:0\n3:0");
  }

  catch (FullErr exception)
  {
    cout << "Error: forest of disjoint sets is full" << endl;
  }

  catch (NotFoundError exception)
  {
    cout << "Error: node not found" << endl;
  }
}

void test_unionSets()
{
  try
  {
    int *a = new int(5);
    int *b = new int(7);
    int *c = new int(20);
    int *d = new int(11);
    int *e = new int(14);
    int *f = new int(22);
    int *g = new int(31);

    DisjointSets<int> set(10);
    DSNode<int> *node1 = set.makeSet(a);
    DSNode<int> *node2 = set.makeSet(b);
    DSNode<int> *node3 = set.makeSet(c);
    DSNode<int> *node4 = set.makeSet(d);
    DSNode<int> *node5 = set.makeSet(e);
    DSNode<int> *node6 = set.makeSet(f);
    DSNode<int> *node7 = set.makeSet(g);

    set.unionSets(node1, node2);
    assert(set.toString() == "5:0 -> 7:1\n7:1\n20:0\n11:0\n14:0\n22:0\n31:0");

    set.unionSets(node1, node3);
    assert(set.toString() == "5:0 -> 7:1\n7:1\n20:0 -> 7:1\n11:0\n14:0\n22:0\n31:0");

    set.unionSets(node4, node5);
    set.unionSets(node5, node6);
    set.unionSets(node6, node7);

    set.unionSets(node1, node2);
    assert(set.toString() == "5:0 -> 7:1 -> 14:2\n7:1 -> 14:2\n20:0 -> 7:1 -> 14:2\n11:0
-> 14:2\n14:2\n22:0 -> 14:2\n31:0 -> 14:2");
  }

  catch (FullErr exception)
  {
    cout << "Error: forest of disjoint sets is full" << endl;
  }

  catch (NotFoundError exception)
  {
    cout << "Error: node not found" << endl;
  }
}
```

```
void test_findSet()
{
  try
  {
    int *a = new int(5);
    int *b = new int(7);
    int *c = new int(20);
    int *d = new int(11);
    int *e = new int(14);
    int *f = new int(22);
    int *g = new int(31);

    DisjointSets<int> set(10);
    DSNode<int> *node1 = set.makeSet(a);
    DSNode<int> *node2 = set.makeSet(b);
    DSNode<int> *node3 = set.makeSet(c);
    DSNode<int> *node4 = set.makeSet(d);
    DSNode<int> *node5 = set.makeSet(e);
    DSNode<int> *node6 = set.makeSet(f);
    DSNode<int> *node7 = set.makeSet(g);

    assert(node6 == set.findSet(node6));

    set.unionSets(node1, node2);
    assert(node2 == set.findSet(node1));
    set.unionSets(node1, node3);
    assert(node2 == set.findSet(node3));

    set.unionSets(node4, node5);
    set.unionSets(node5, node6);
    set.unionSets(node6, node7);
    assert(node5 == set.findSet(node6));
    assert(node5 == set.findSet(node7));

    set.unionSets(node1, node7);
    assert(node5 == set.findSet(node1));
    assert(node5 == set.findSet(node2));
    assert(node5 == set.findSet(node3));
  }

  catch (FullErr exception)
  {
    cout << "Error: forest of disjoint sets is full" << endl;
  }

  catch (NotFoundError exception)
  {
    cout << "Error: node not found" << endl;
  }
}

int main()
{
  test_constructor();
  test_makeSet();
  test_unionSets();
  test_findSet();

  return 0;
}
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

#ifndef LIST_H
#define LIST_H

#include <cstdlib>
#include <iostream>

template <class T>
class Node
{
 public:

   T *item;
   Node<T> *next;

   Node();
   Node(T *initItem);
   Node(T *initItem, Node<T> *initNext);
};

template <class T> class List;

template <class T>
std::ostream& operator<<(std::ostream& os, const List<T>& list);

template <class T>
class List
{
 public:

   List();                                  // default constructor
   List(const List<T>& src);                // copy constructor
   ~List();                                 // destructor

   void append(T *item);                    // append a new item to the end of the list
   int length() const;                      // return the number of items in the list
   int index(const T& item) const;          // return index of value item, or -1 if not foun
d
   void insert(int index, T *item);         // insert item in position index
   T *pop(int index);                       // delete the item in position index and return
it
   void remove(const T& item);              // remove the first occurrence of the value item

   T *operator[](int index);                      // indexing operator
   List<T>& operator=(const List<T>& src);        // assignment operator
   List<T>& operator+=(const List<T>& src);       // concatenation operator

 private:

   Node<T> *head;                           // head of the linked list
   int count;                               // number of items in the list

   void deepCopy(const List<T>& src);
   void deallocate();                       // deallocate the list
   Node<T>* _find(int index);               // return a pointer to the node in position inde
x

   friend std::ostream& operator<< <T>(std::ostream& os, const List<T>& list);
};

class IndexError { };
class ValueError { };

#include "list.cpp"

#endif
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

template <class T>
Node<T>::Node()
{
   item = NULL;
   next = NULL;
}

template <class T>
Node<T>::Node(T *initItem)
{
   item = initItem;
   next = NULL;
}

template <class T>
Node<T>::Node(T *initItem, Node<T> *initNext)
{
   item = initItem;
   next = initNext;
}

template <class T>
List<T>::List()
{
        head = NULL;
        count = 0;
}

template <class T>
List<T>::List(const List<T>& src)
{
        deepCopy(src);
}

template <class T>
List<T>::~List()
{
        deallocate();
}

template <class T>
List<T>& List<T>::operator=(const List<T>& src)
{
        deallocate();
        deepCopy(src);

        return *this;
}

template <class T>
int List<T>::length() const
{
        return count;
}

template <class T>
int List<T>::index(const T& item) const
{
        int index = 0;
        Node<T> *node = head;
        while ((node != NULL) && (!(*(node->item) == item)))
        {
                node = node->next;
                index++;
        }
```

```
        if (node == NULL)
                return -1;
        else
                return index;
}

template <class T>
void List<T>::append(T *item)
{
        Node<T> *node,
                *newNode;

        newNode = new Node<T>(item);

        if (head != NULL)
        {
                node = _find(count - 1);
                node->next = newNode;
        }
        else
                head = newNode;

        count++;
}

template <class T>
void List<T>::insert(int index, T *item)
{
        if ((index < 0) || (index > count))
                throw IndexError();

        Node<T> *node;

        if (index == 0)
                head = new Node<T>(item, head);
        else
        {
                node = _find(index - 1);
                node->next = new Node<T>(item, node->next);
        }
        count++;
}

template <class T>
T *List<T>::pop(int index)
{
        if ((index < -1) || (index >= count))
                throw IndexError();

        if (index == -1)
                index = count - 1;

        Node<T> *node, *dnode;
        T *item;

        if (index == 0)
        {
                dnode = head;
                head = head->next;
                item = dnode->item;
                delete dnode;
        }
        else
        {
                node = _find(index - 1);
                if (node != NULL)
                {
                        dnode = node->next;
                        node->next = node->next->next;
```

```
                                item = dnode->item;
                                delete dnode;
                        }
                }
                count --;
                return item;
}

template <class T>
T* List<T>::operator[](int index)
{
        if ((index < 0) || (index >= count))
                throw IndexError();

        Node<T> *node = _find(index);
        return node->item;
}

template <class T>
void List<T>::deepCopy(const List<T>& src)
{
        Node<T> *snode, *node;

        snode = src.head;
        if (snode != NULL)
        {
                node = head = new Node<T>(snode->item);
                snode = snode->next;
        }
        while (snode != NULL)
        {
                node->next = new Node<T>(snode->item);
                node = node->next;
                snode = snode->next;
        }

        count = src.count;
}

template <class T>
void List<T>::deallocate()
{
        Node<T> *node, *dnode;

        node = head;
        while (node != NULL)
        {
                dnode = node;
                node = node->next;
                delete dnode;
        }
}

template <class T>
void List<T>::remove(const T& item)
{
        if (head == NULL)
                return;

        Node<T> *toDelete;

        if (*(head->item) == item)
        {
                toDelete = head;
                head = head->next;
                delete toDelete;
                count--;
        }
        else
        {
```

```
          Node<T> *node = head;
          while ((node->next != NULL) && (!(*(node->next->item) == item)))
                  node = node->next;

          if (node->next != NULL)
          {
                  toDelete = node->next;
                  node->next = node->next->next;
                  delete toDelete;
                  count--;
          }
        }
}

template <class T>
Node<T>* List<T>::_find(int index)  // used by append, insert, [], pop
{
        if ((index < 0) || (index >= count))
                throw IndexError();

        Node<T> *node = head;
        for (int i = 0; i < index; i++)
                node = node->next;

        return node;
}

template <class T>
std::ostream& operator<<(std::ostream& os, const List<T>& list)
{
        Node<T> *node = list.head;

        while (node != NULL)
        {
                os << *(node->item) << " ";
                node = node->next;
        }
        os << std::endl;

        return os;
}
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

#ifndef GRAPH
#define GRAPH

#include <iostream>
#include <string>
#include "DSF.h"
#include "list.h"
#include "pq.h"

using namespace std;

class Vertex
{
public:
  Vertex(); // default constructor
  int ident;
  string color; // vertex attributes
  int finish;
  int dist;
  Vertex *pred; // predecessor vertex
};

class Pair
{
public:
  Pair(); // default constructor
  Pair(Vertex *initV1, Vertex *initV2, int initW); // constructor
  bool operator<(const Pair& p) const; // overloading < operator
  Vertex *v1; // vertex
  Vertex *v2; // vertex that is connected to v1
  int w; // weight of vertex inside pair
};

template<class T>
class Graph
{
public:
  Graph(string filename); // default constructor that reads a graph from a file
  Graph(const Graph<T>& gra); // copy constructor
  ~Graph(); // destructor
  void DFS(); // Depth-First Search (search all of the vertices)
  void Kruskal(); // An algorithm to find a minimum spanning tree

private:
  int time; // used for DFS(). Record the time
  Vertex **vertices; // array of vertices read from a file
  Pair *pairElem; // pointer to a Pair object
  List<Pair> **adjElements; // array pointing to a list of pairs
  int count; // count
  int countE; // number of edges
  void DFS_Visit(Vertex *u); // Depth-First Search method to change color of visited vert
ex
  std::string toString() const; // toString method
};

template<class T>
std::ostream& operator<<(std::ostream& stream, const Graph<T>& graf); // ostream operator

#include "Graph.cpp"
#endif
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

#include <iostream>
#include <string>
#include <fstream>
#include <stdlib.h>
#include <sstream>

using namespace std;

/*=======================================================
Vertex Default Constructor
Precondition: None
Postcondition: Initializes generic Vertex object
=======================================================*/
Vertex::Vertex()
{
  ident = 0;
  finish = 0;
  dist = 0;
  color = "White";
  pred = NULL;
}

/*=======================================================
Pair Default Constructor
Precondition: None
Postcondition: Initializes empty Pair object
=======================================================*/
Pair::Pair()
{
  v1 = NULL;
  v2 = NULL;
  w = 0;
}

/*=======================================================================
Pair Constructor
Precondition: initV1 and initV2 are pointers to Vertices, initW is an integer
Postcondition: creates pair with vertices initV1 and initV2 and weight initW
=======================================================================*/
Pair::Pair(Vertex *initV1, Vertex *initV2, int initW)
{
  v1 = initV1;
  v2 = initV2;
  w = initW;
}

/*=======================================================================
Pair Overloading Operator
Precondition: p is a Pair object
Postcondition: returns true if p has greater weight, false otherwise
=======================================================================*/
bool Pair::operator<(const Pair& p) const
{
  if(this->w < p.w)
    return true;
  else
    return false;
}

/*==========================================================================
Graph Default Constructor
Precondition: text file with name filename exists, file filename contains a line
with an integer n followed by n rows containing n elements each
Postcondition: initializes graph based on adjacency matrix from filename
==========================================================================*/
```

```
template<class T>
Graph<T>::Graph(string filename)
{
  count = 0;
  countE = 0;
  char ch; // initializing graph object
  string line;
  int identNumb = 0;

  ifstream infile; // file to read from
  infile.open(filename.c_str()); // filename we are reading from
  getline(infile, line); // getline to get number of vertices
  int s = atoi(line.c_str()); // converting number of vertices from txt file to an int
  count = s;
  vertices = new Vertex*[s]; // dynamically allocates an array of pointer to vertices

  for(int v = 0; v < s; v++)
  {
    vertices[v] = new Vertex;
    vertices[v]->ident = v; // giving each vertex and identifcation number from 0 to n-1
  }

  adjElements = new List<Pair>*[s]; // dynamically allocates an array that points to list
 of pairs

  for(int i = 0; i < s; i++)
  {
    int k = 0;
    adjElements[i] = new List<Pair>; // creating list of pairs in each array slot
    getline(infile, line);

    for(int j = 0; j < line.length(); j++)
    {
      if(line[j] != '0' && line[j] != ' ')
      {
        int intLine = line[j] - 48;
        pairElem = new Pair(vertices[i], vertices[k], intLine); // dynamically allocating
 a pair
        // object that contains vertex, the vertex it is connected to, and the edge weigh
t
        adjElements[i]->append(pairElem); // appending pair object to list of pairs in ar
ray
        countE++; // incrementing count
      }
      if(line[j] != ' ')
        k++;
    }
  }
}

/*=============================================================================
Graph Copy Constructor
Precondition: gra is a Graph object
Postcondition: initializes a Graph as a copy of gra
=============================================================================*/
template<class T>
Graph<T>::Graph(const Graph<T>& gra)
{
  count = gra.count;
  countE = gra.countE;
  time = gra.time;
  pairElem = gra.pairElem;
  vertices = gra.vertices;
  adjElements = new List<Pair>*[count];

  for(int i = 0; i < count; i++)
  {
    adjElements[i] = gra.adjElements[i];
  }
}
```

```
/*============================================================================
Graph Destructor
Precondition: None
Postcondition: Deallocates memory of the graph
============================================================================*/
template<class T>
Graph<T>::~Graph()
{
  for(int i = 0; i < count; i++)
  {
    List<Pair> tempPairList = *(adjElements[i]);
    for(int j = 0; j < adjElements[i]->length(); j++)
    {
      delete tempPairList[j];
    }
  }
}

/*==================================================================================
Graph Depth-First Search
Precondition: There is a Graph object
Postcondition: prints identifiers of vertices in the order they are visited in a DFS
==================================================================================*/
template<class T>
void Graph<T>::DFS()
{
  Vertex *u;
  for(int i = 0; i < count; i++)
  {
    // initializes vertex u, local variable
    u = vertices[i];
    u->color = "White";
    u->pred = NULL;
  }
  time = 0;
  for(int i = 0; i < count; i++)
  {
    u = vertices[i];
    if(u->color == "White")
    {
      DFS_Visit(u); // calls DFS_Visit recursively
    }
  }
}

/*============================================================================
Depth-First Search method to change color of visited vertex
Precondition: u points to a vertex in the graph
Postcondition: updates time and finish of all vertices, prints identifier of U
============================================================================*/
template<class T>
void Graph<T>::DFS_Visit(Vertex *u)
{
  Vertex *v; // local vertex
  u->color = "Gray"; // setting to "Gray", indicates visited
  time++;
  u->dist = time;
  List<Pair> temp = *(adjElements[u->ident]); // obtains adjacency list of u
  cout << u->ident << " Visit Time -> " << time << endl;

  for(int i = 0; i < temp.length(); i++)
  {
    v = temp[i]->v2;
    if(v->color == "White")
    {
      v->color = "Gray";
      v->pred = u;
      DFS_Visit(v);
    }
```

```
  }

  u->color = "Black"; // setting to "Black", indicates finished
  time++;
  u->finish = time;
  cout << u->ident << " Finish Time -> " << time << endl;
}

/*================================================================================
Kruskal Minimum Spanning Tree algorithm
Precondition: There is a Graph object
Postcondition: prints edges in the order they are added to a spanning tree during
Kruskal's algorithm
================================================================================*/
template<class T>
void Graph<T>::Kruskal()
{
  Vertex *Vex; // local vertex
  Pair VexPair; // pair object
  DSNode<Vertex>* VexNode; // pointer to a DSNode that contains a vertex
  DisjointSets<Vertex> DSets; // disjoint set containing vertices
  MinPriorityQueue<Pair> weightsQueue(countE); // mpq that contains pairs with capacity c
ountE
  List< DSNode<Vertex> > VexList; // list that holds DSNodes that contain vertices

  for(int i = 0; i < count; i++)
  {
    Vertex* Vex = new Vertex; // dynamically allocates pointer to a vertex
    Vex = vertices[i]; // gets each vertex from list of vertices
    VexNode = DSets.makeSet(Vex); // makeSet called on each vertex and as signed to DSNod
e holding vertices
    VexList.append(VexNode); // appends DSNode holding vertices from makeSet to a list
  }

  for(int j = 0; j < count; j++)
  {
    List<Pair> tempList = *(adjElements[j]); // list of pairs representing adjacency of e
ach certain vertex
    for(int k = 0; k < adjElements[j]->length(); k++)
    {
      weightsQueue.insert(tempList[k]); // inserting the adjacency nodes to MPQ
    }
  }

  cout << "[ ";
  for(int i = 0; i < countE; i++)
  {
    Pair *shortEdge = weightsQueue.extractMin(); // extracts min to get smallest weights
    DSNode<Vertex>* U = VexList[shortEdge->v1->ident]; // DSNodes containing vertex v1
    DSNode<Vertex>* V = VexList[shortEdge->v2->ident]; // DSNodes containing vertex v2 co
nnected to v1

    if(DSets.findSet(U) != DSets.findSet(V))
    {
      DSets.unionSets(U, V); // calls unionSets to both vertics
      cout << "{" << U->key->ident << "," << V->key->ident << "} ";
    }
  }
  cout << "]" << endl;
}

/*============================================================
Graph toString method
Precondition: There is a Graph object
Postcondition: returns a string representation of the Graph
============================================================*/
template<class T>
string Graph<T>::toString() const
{
  stringstream s;
```

```
  for(int i = 0; i < count; i++)
  {
    List<Pair> tempList = *(adjElements[i]); // list of pairs representing adjacency of e
ach certain vertex
    for(int j = 0; j < adjElements[i]->length(); j++)
    {
      Pair pr = pairElem[j]; // creating pair of each element pairElem is pointing to
      s << i << " -> " << tempList[j]->v2->ident << " with weight: " << tempList[j]->w <<
 ", " << tempList[j]->v2->color << endl;
    }
  }

  return s.str();
}

/*=====================================================
ostream operator
Precondition: graf is a Graph object
Postcondition: prints a string representation of graf
=====================================================*/
template<class T>
ostream& operator<<(std::ostream& stream, const Graph<T>& graf)
{
  stream << graf.toString();
  return stream;
}
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

#include "Graph.h"

int main()
{
  Graph<Vertex> lejames("lejames.txt");
  lejames.DFS();
  lejames.Kruskal();
  Graph<Vertex> lejames2(lejames);
  return 0;
}
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

// pq.h
// This MinPriorityQueue template class assumes that the class KeyType has
// overloaded the < operator and the << stream operator.

#ifndef PQ_H
#define PQ_H

#include <iostream>
#include "heap.h"

template <class KeyType>
class MinPriorityQueue : public MinHeap<KeyType>
{
  public:
    MinPriorityQueue();              // default constructor
    MinPriorityQueue(int n);         // construct an empty MPQ with capacity n
    MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);  // copy constructor

    KeyType* minimum() const;                    // return the minimum element
    KeyType* extractMin();                       // delete the minimum element and return
it
    void decreaseKey(int index, KeyType* key);  // decrease the value of an element
    void insert(KeyType* key);                   // insert a new element
    bool empty() const;                          // return whether the MPQ is empty
    int length() const;                          // return the number of keys
    std::string toString() const;                // return a string representation of the
MPQ

    // Specify that MPQ will be referring to the following members of MinHeap<KeyType>.

    using MinHeap<KeyType>::A;
    using MinHeap<KeyType>::heapSize;
    using MinHeap<KeyType>::capacity;
    using MinHeap<KeyType>::parent;
    using MinHeap<KeyType>::swap;
    using MinHeap<KeyType>::heapify;

    /* The using statements are necessary to resolve ambiguity because
       these members do not refer to KeyType.  Alternatively, you could
       use this->heapify(0) or MinHeap<KeyType>::heapify(0).
    */
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq);

class FullError { };     // MinPriorityQueue full exception
class EmptyError { };    // MinPriorityQueue empty exception
class KeyError { };      // MinPriorityQueue key exception

#include "pq.cpp"

#endif
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill
// pq.cpp

#include <iostream>
#include <string>
#include <sstream>
#include <cstdlib>
#include <cassert>

using namespace std;

/*==================================================
MinPriorityQueue()              // default constructor
Precondition: None
Postcondition: An empty priority queue
==================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue()
{
        //capacity = 100;
        heapSize = 0 ;
        A = new KeyType* [capacity];
}

/*======================================================================
MinPriorityQueue(int n)         // construct an empty MPQ with capacity n
Precondition: Must be given a capacity size (n)
Postcondition: An empty priority queue with capacity of n
======================================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(int n)
{
        capacity = n;
        heapSize = 0;
        A = new KeyType* [n];
}

/*=========================================================================
MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);  // copy constructor
Precondition: Must be given a priority queue pq
Postcondition: Traverses the priority queue and makes a copy of its values
        to transfer to another priority queue
=========================================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(const MinPriorityQueue<KeyType>& pq)
{
        heapSize = pq.heapSize;
        capacity = pq.capacity;
        A = new KeyType*[capacity];

        for (int i=0; i < heapSize; i++){
                A[i] = pq[i];
        }

        // buildHeap()
        heapSize = capacity;
        for (int i = (capacity/2); i >= 0; i--)
                heapify(i);
}

/*=================================================================
KeyType* minimum() const                // return the minimum element
Precondition: A non-empty min-heap A
Postcondition: Returns the minimum value in min-heap A
=================================================================*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::minimum() const
```

```
{
        if (empty())
                throw EmptyError();
        return A[0];
}

/*=============================================================================
KeyType* extractMin()            // delete the minimum element and return it
Precondition: A non-empty min-heap A
Postcondition: Deletes the minimum value in min-heap A and returns it
=============================================================================*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::extractMin()
{
        if (empty())
                throw EmptyError();
        KeyType* min = (A[0]);
        A[0] = A[heapSize-1];
        heapSize--;
        heapify(0);
        return min;
}

/*==============================================================================
void decreaseKey(int index, KeyType* key)  // decrease the value of an element
Precondition: A min-heap A where new key is always smaller than current key
Postcondition: The value of element index's key has the new value key
==============================================================================*/
template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey(int index, KeyType* key)
{
        if (*(A[index]) < *key)
                throw KeyError();
        A[index] = key;
        while ((index > 0) && (*(A[index]) < *(A[parent(index)]))) {
                swap(index, parent(index));
                index = parent(index);
        }
}

/*===============================================================================
void insert(KeyType* key)                  // insert a new element
Precondition: Input is the key of the new element to be inserted into min-heap A
Postcondition: Key of the new node is in correct value and the heap maintains
        its min-heap property
===============================================================================*/
template <class KeyType>
void MinPriorityQueue<KeyType>::insert(KeyType* key)
{
        if (heapSize == capacity)
                throw FullError();

        if(heapSize ==0){
                A[heapSize] = key;
                heapSize++;
        }
        else{
        A[heapSize] = key;
        decreaseKey(heapSize, key);
        heapSize++;
        }
}


/*============================================================================
bool empty() const                         // return whether the MPQ is empty
Precondition: None
Postcondition: Returns true if the priority queue is empty, false otherwise
============================================================================*/
template <class KeyType>
```

```
bool MinPriorityQueue<KeyType>::empty() const
{
        if (heapSize == 0)
                return 1;
        else
                return 0;
}


/*=======================================================
int length() const         // return the number of keys
Precondition: None
Postcondition: Returns the length of the priority queue
=======================================================*/
template <class KeyType>
int MinPriorityQueue<KeyType>::length() const
{
        return heapSize;
}


/*==================================================================================
std::string toString() const          // return a string representation of the MPQ
Precondition: A priority queue to be converted to a string
Postcondition: Traverses the array and uses << to output each element of the array
==================================================================================*/
template <class KeyType>
std::string MinPriorityQueue<KeyType>::toString() const
{
        stringstream result; //sets variable to be returned
        int x = 0;
        result << "[";
        int size = heapSize;

        while(x < size) // inserts values into "result" while traversing list
        {
                result << *(A[x]);
                x++;
                if(x != size)
                        result << ",";
        }
        result << "]";
        return result.str();
}

/*==================================================================================
std::string toString() const          // return a string representation of the MPQ
Precondition: A priority queue to be converted to a string
Postcondition: Traverses the array and uses << to output each element of the array
==================================================================================*/
template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq)
{
        stream << pq.toString();
  return stream;
}
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

// heap.h
// a binary min heap

#ifndef HEAP_H
#define HEAP_H

#include <iostream>

const int DEFAULT_SIZE = 100;

template <class KeyType>
class MinHeap
{
  public:
    MinHeap(int n = DEFAULT_SIZE);           // default constructor
    MinHeap(KeyType* initA[], int n);         // construct heap from array
    MinHeap(const MinHeap<KeyType>& heap);  // copy constructor
    ~MinHeap();                              // destructor

    void heapSort(KeyType* sorted[]);  // heapsort, return result in sorted

    MinHeap<KeyType>& operator=(const MinHeap<KeyType>& heap);  // assignment operator
    std::string toString() const;     // return string representation

  protected:
    KeyType **A;     // array containing the heap
    int heapSize;   // size of the heap
    int capacity;   // size of A

        void heapify(int index);           // heapify subheap rooted at index
    void buildHeap();                  // build heap
        int leftChild(int index) { return 2 * index + 1; }  // return index of left child
        int rightChild(int index) { return 2 * index + 2; } // return index of right chil
d
        int parent(int index) { return (index - 1) / 2; }   // return index of parent
    void swap(int index1, int index2);        // swap elements in A
    void copy(const MinHeap<KeyType>& heap);  // copy heap to this heap
    void destroy();                           // deallocate heap
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinHeap<KeyType>& heap);

#include "heap.cpp"

#endif
```

```
// James Le
// Project 1000
// CS 271: Data Structure
// Dr. Jessen Havill

// heap.cpp

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <string>
#include <sstream>
#include <cassert>

using namespace std;
/*==============================================================
MinHeap(int n = DEFAULT_SIZE)      //default constructor
Precondition: Must be given a capacity size (n)
Postcondition: An empty heap with capacity of n (1000 (default))
==============================================================*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(int n)
{
        heapSize = 0;
        capacity = n;
        A = new KeyType*[n];
}

/*==============================================================
MinHeap(KeyType initA[], int n)    //construct heap from array
Precondition: Must be given an array and capacity
Postcondition: Traverses and makes a copy of the array. Then creates a min-heap
                using buildHeap
==============================================================*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(KeyType* initA[], int n)
{
        A = new KeyType*[n];
        for (int i=0; i < n; i++){
                A[i] = initA[i];
        }
        capacity = n;
        buildHeap();
}

/*==============================================================
MinHeap(const MinHeap<KeyType>& heap);  // copy constructor
Precondition: Must be given a heap
Postcondition: Traverses the heap and makes a copy of its values
                to transfer to another heap
==============================================================*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(const MinHeap<KeyType>& heap)
{
        heapSize = heap.heapSize;
        capacity = heap.capacity;
        A = new KeyType*[capacity];
        for (int i=0; i < heapSize; i++){
                A[i] = heap[i];
        }
        buildHeap();

}

/*==============================================================
 ~MinHeap();                               // destructor
Precondition: N/A
Postcondition: deallocates the heap
==============================================================*/
template <class KeyType>
```

```
MinHeap<KeyType>::~MinHeap()
{
        delete []A;
}

/*============================================================
heapSort(KeyType sorted[]);  // heapsort, return result in sorted
Precondition: Must be given a heap to be sorted
Postcondition: Uses the min-heap property to continuously insert
                the next smallest value of a decreasing heap in
                order to sort it. A sorted (ascending) heap is returned
=============================================================*/
template <class KeyType>
void MinHeap<KeyType>::heapSort(KeyType* sorted[])
{
        int temp;
        int var = heapSize;

        for(int i=0; i < var; i++){
                sorted[i] = A[0];
                swap(0, heapSize-1);

                heapSize--;
                heapify(0);
        }


        for(int j=0; j < var; j++){
                A[j] = sorted[j];
        }

        heapSize = var;
        return;


}


/*============================================================
operator = (const MinHeap<KeyType>& heap);  // assignment operator
Precondition: Must be given a heap to be copied
Postcondition: Copies the heap size and capacity of a heap, then
                traverses it to copy the elements into another heap
=============================================================*/
template <class KeyType>
MinHeap<KeyType>& MinHeap<KeyType>::operator=(const MinHeap<KeyType>& heap)
{

        heapSize = heap.heapSize;
        capacity = heap.capacity;

        for(int i=0; i < heapSize; i++)
                A[i] = heap.A[i];

        copy(heap);
}


/*============================================================
operator << (std::ostream& stream, const MinHeap<KeyType>& heap)
Precondition: Must be given a heap
Postcondition: Calls the function toString and returns the output
=============================================================*/
template <class KeyType>
std::ostream& operator<< (std::ostream& stream, const MinHeap<KeyType>& heap)
{
        stream << heap.toString();
        return stream;
}
```

```
/*==============================================================
toString() const;      // return string representation
Precondition: A heap to be converted to a string (not as a parameter)
Postcondition: Traverses the array and uses << to output each element of
                the array
==============================================================*/
template <class KeyType>
std::string MinHeap<KeyType>::toString() const
{
        stringstream result; //sets variable to be returned
        int x = 0;
        result << "[";
        int size = heapSize;

        while(x < size) // inserts values into "result" while traversing list
        {
                result << *(A[x]);
                x++;
                if(x != size)
                        result << ",";
        }
        result << "]";
        return result.str();
}



/*==============================================================
heapify(int index);           // heapify subheap rooted at index
Precondition: Must be given an index. Used on an array
Postcondition: Recursively compares the parent, left child, and
                right child. Places the smaller of the three values
                in the index.
==============================================================*/
template <class KeyType>
void MinHeap<KeyType>::heapify(int index)
{
        int smallest;
        int l = leftChild(index);
        int r = rightChild(index);
        if (l < heapSize and *(A[l]) < *(A[index]))
                smallest = l;
        else
                smallest = index;

        if (r < heapSize and *(A[r]) < *(A[smallest]))
                smallest = r;
        if (smallest != index){
                swap(index,smallest);
                heapify(smallest);
        }
        return;
}

/*==============================================================
buildHeap();                  // build heap
Precondition: Used on a heap
Postcondition: Traverses half of the array and calls heapify(i)
==============================================================*/
template <class KeyType>
void MinHeap<KeyType>::buildHeap()
{
        heapSize = capacity;
        for (int i = (capacity/2); i >= 0; i--)
                heapify(i);

}
```

```
/*===========================================================
swap(int index1, int index2);        // swap elements in A
Precondition: Must be given two indices
Postcondition: Exchanges the values of the two indices
=============================================================*/
template <class KeyType>
void MinHeap<KeyType>::swap(int index1, int index2)
{
        KeyType* temp = A[index1];
        A[index1] = A[index2];
        A[index2] = temp;

}

/*===========================================================
copy(const MinHeap<KeyType>& heap);  // copy heap to this heap
Precondition: Must be given a heap to copy
Postcondition: Calls the copy constructor
=============================================================*/
template <class KeyType>
void MinHeap<KeyType>::copy(const MinHeap<KeyType>& heap)
{

        MinHeap(A);

}

/*===========================================================
destroy();                           // deallocate heap
Precondition: N/A
Postcondition: Calls the deconstructor
=============================================================*/
template <class KeyType>
void MinHeap<KeyType>::destroy()
{
        ~MinHeap();

}
```